PerDiS: design, implementation, and use of a PERsistent DIstributed Store

Paulo Ferreira, Marc Shapiro, Xavier Blondel, Olivier Fambon, João Garcia, Sytse Kloosterman, Nicolas Richer, Marcus Roberts, Fadi Sandakly, George Coulouris, et al.

PerDiS: design, implementation, and use of a PERsistent DIstributed Store *

Paulo Ferreira[1], Marc Shapiro[2], Xavier Blondel[2**], Olivier Fambon[3], João Garcia[1***], Sytse Kloosterman[2], Nicolas Richer[2†], Marcus Robert[4], Fadi Sandakly[5], George Coulouris[4], Jean Dollimore[4], Paulo Guedes[1], Daniel Hagimont[3], and Sacha Krakowiak[3]

[1] INESC
Rua Alves Redol 9, 1000 Lisboa, Portugal
{paulo.ferreira,joao.c.garcia,paulo.guedes}@inesc.pt
[2] INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt BP 105, Le Chesnay Cedex 78153, France
{marc.shapiro,xavier.blondel,sytse.kloosterman,nicolas.richer}@inria.fr
[3] INRIA Rhône-Alpes
ZIRST - 655, Avenue de l'Europe, 38330 Montbonnot Saint-Martin, France
{olivier.fambon,daniel.hagimont,sacha.krakowiak}@inrialpes.fr
[4] QMW
Queen Mary and Westefield College, Mile End Road, London, E1 4NS, UK
{marcusr,george,jean}@dcs.qmw.ac.uk
[5] CSTB
BP 209, 290 Route de Lucioles, F-06904 Sophia Antipolis Cedex, France
sandakly@cstb.fr

Abstract The PerDiS (Persistent Distributed Store) project addresses the issue of providing support for distributed collaborative engineering applications. We describe the design and implementation of the PerDiS platform, and its support for such applications.
Collaborative engineering raises system issues related to the sharing of large volumes of fine-grain, complex objects across wide-area networks and administrative boundaries. PerDiS manages all these aspects in a well defined, integrated, and automatic way. Distributed application programming is simplified because it uses the same memory abstraction as in the centralized case. Porting an existing centralized program written in C or C++ is usually a matter of a few, well-isolated changes.
We present some performance results from a proof-of-concept platform that runs a number of small, but real, distributed applications on Unix

* This work was supported by Esprit under the PerDiS project (n^o 22533), http://www.perdis.esprit.ec.org/. The partners in the PerDiS consortium are: CSTB, INRIA Rhône-Alpes, INESC, QMW College, INRIA Rocquencourt, IEZ. Two are from the building industry: CSTB is a research institute, and IEZ is a CAD tool company. The others are research institutes or university departments in Informatics and Telecommunications. The Principal Investigator is Marc Shapiro.

** Student at CNAM Cédric.

*** Student at IST, Technical University of Lisbon.

† Student at Université Paris-6, LIP6.

and Windows NT. These confirm that the PerDiS abstraction is well adapted to the targeted application area and that the overall performance is promising compared to alternative approaches.

1 Introduction

The PerDiS project seeks to support distributed, cooperative engineering applications in the large scale. It aims to demonstrate cooperative computer-aided design (CAD) of buildings within a virtual enterprise.[1]

Single-user CAD applications are in widespread use today in architecture and building firms. The design for a building contains numerous fine-grain objects (100 bytes–10 Kb each), typically running into megabytes even for a relatively simple building. Objects are densely interconnected by pointers; for instance a wall object contains a pointer to its adjacent walls, ceiling, and floor, as well as to its windows, doors, pipes and other fittings.

In current practice, sharing of information in a VE is mostly limited to faxes or sending diskettes by post.[2] On a smaller scale, an enterprise might share files through a distributed file system over a local network, but is hindered in this case by the lack of consistency and concurrency control.

The industrial demand for distributed, collaborative CAD tools is high. Many developments, including one by PerDiS partner CSTB [1], are based on remote object invocation, using Corba [16], DCOM [29] or Java RMI [34]. A client application running on a workstation invokes objects, stored in a server, through remote references. Applied to the CAD domain this results in abysmal performance, and server scalability problems. Remote objects are especially inappropriate in the virtual enterprise, where the object server may be located across a slow WAN connection. Applications must be completely re-engineered (in devious ways) in order to get decent performance. Furthermore, none of the remote-object systems adequately address persistence or concurrency control.

Collaborative engineering in a VE raises a number of exciting system issues. The goal of this research is to address them in a fully integrated, automated, efficient and easy-to-use platform. Application programmers should be able to concentrate on application semantics, without worrying about system issues. Existing centralized CAD applications must port easily without complete re-engineering. The platform should provide fast, consistent access to data, despite concurrent access. Persistence must be guaranteed for as long as objects are needed. The platform should automate distribution, storage, and input-output. The system should work well in the large scale, tolerating faults such as network slowdowns and disconnections, crashes, and providing an adequate level of security.

[1] A virtual enterprise (VE) is a consortium of small enterprises, or of small departments of larger enterprises, working together for the duration of a construction project. The members are often geographically dispersed, even located in different countries.

[2] Users are limited not only by the absence of interchange standards or of high-speed networks, but also by a prudent distrust between members of the VE.

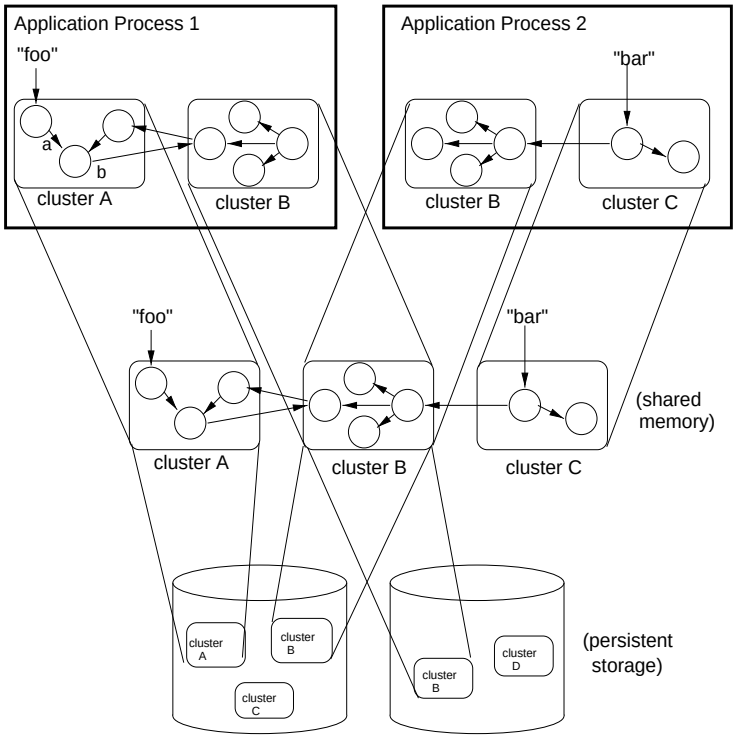

Figure1. PerDiS abstractions

In a VE, collaboration follows a stylized, sequential pattern. Typically, a small group of architects located at a single site do the initial design, performing many updates during a limited period of time. Then, the design is passed along to structural engineers, another small group, possibly in a different site. They then pass their results on to another group, and so on. There is a high degree of temporal and spatial locality. There is also some real concurrency, including write conflicts, which cannot be ignored; for instance working on alternative designs in parallel is common practice.

To better understand the PerDiS approach, consider a typical application scenario from the building industry area. Architects and engineers from different companies collaborate on a design, working at different locations, either concurrently or at different times. Tentative or alternative designs are created, tried out, abandoned. The constructors on site consult the plans, making on-the-spot modifications, which should be reflected back to the engineering offices.

In response to these requirements, the PerDiS project proposes a new abstraction, the *Persistent Distributed Store*.

This paper is organized as follows. In the next section we present the concepts of PerDiS. Section 3 describes system's layers of functionality. Section 4

describes the architecture and implementation of the PerDiS platform. In Section 5 we show how application programmers use PerDiS. We report results of some experiments in Section 7. Section 8 compares our approach with related work. We conclude in Sections 9 and 10 with lessons learned and future plans.

2 PerDiS Concepts

Figure 1 presents the conceptual model of a persistent distributed store. An application process maps a distributed, shared, persistent memory. It accesses this memory transactionally. The memory is divided into clusters, containing objects. Named roots provide the entry points. Objects are connected by pointers. Reachable objects are stored persistently in clusters on disk; unreachable objects are garbage-collected.

PerDiS provides direct, in-memory access to distributed and persistent objects. Application programmers concentrate on application development without being distracted by system issues. Moreover, knowledgeable programmers have full control over distribution and concurrency control.

Shared address space PerDiS supports the *Shared Address Space* model [11]. It is simple, natural and easy to use, because it provides the same, familiar, memory abstraction as in the centralized case. It facilitates the sharing of data between programs, just by naming, assigning and dereferencing pointers.

PerDiS provides the illusion of a shared memory across the network and across time, by transparently and coherently caching data in memory and reading and writing to disk.

Clusters An application allocates an object within a cluster of the shared memory. A cluster groups together objects that belong together for locality, concurrency control, garbage collection and protection.

The intent is that a cluster will be used just like a document or a file in current operating systems, providing the user with mnemonic access to important data. Applications divide their data among clusters in whatever way is most natural and provides best locality. For example, in our cooperative engineering applications, objects for each major section of a building will be stored in a separate cluster.

Persistence by reachability The PerDiS memory is persistent. Even programs running at different times share data simply by mapping memory and working with pointers. The application programmer does not need to worry about flattening object graphs into files, nor about parsing ASCII representations into memory.

An object may point to any other object. The system ensures that pointers retain the same meaning for all application processes. To combine persistence with real pointers while retaining flexibility, many systems do *swizzling*, i.e., automatically translate global addresses into pointers [27,33]. PerDiS is designed to

provide swizzling but the current implementation simply relies on fixed address allocation [11].

Starting from some *persistent root* pointer identified by a string name, a process navigates from object to object by following pointers. For instance, an application might navigate the graph illustrated in Figure 1. Starting from the root `foo` in cluster A, one can navigate through the objects, e.g. invoke `foo->a->b->print()`. This would access cluster B.

Any object reachable through some path from a persistent root must persist on permanent storage. This is called *Persistence By Reachability* (PBR) [3]. Unreachable objects are garbage-collected (see Section 4.4).

Transactions and concurrency control To relieve application programmers from dealing with the hard issues of concurrent updates, an application runs as a sequence of one or more transactions. It can read and write memory without interference from concurrently-running processes. Transactions ensure that if a single transaction updates multiple clusters, either the transaction commits and all the updates are applied, or it aborts and it is as if none has occurred.

For completeness, PerDiS supports transactions with the usual ACID transactional semantics. However, the ACID model is not well suited to our application area, so we plan to support more sophisticated transactions that allow more concurrency while reducing the probability of aborts.

Security A VE is a co-operation between different companies, for the limited purpose of designing and constructing some building. These same companies might be simultaneously competing for some other building. Data and communication must be protected.

PerDiS poses new security problems because applications access local copies of objects (via a DSM mechanism) instead of accessing them as remote objects protected by servers (more details in Section 4.5).

Ease of programming It is a requirement to minimize programming restrictions, and to support standard, non-modified programming languages and compilers. Pointers in C and C++ (or even assembly language) are supported; there is no requirement to use special pointer types (e.g., C++ smart pointers [17]), and dereferencing a pointer costs the same (once data has been loaded) as in the machine's native virtual memory. Pointer arithmetic is legal. However, pointer-hiding (e.g., XORing pointers, casting a non-pointer into a pointer, or a union of a pointer and a non-pointer) would defeat garbage collection and is illegal.

Our performance goals are modest: within the limits of our locality model (see Section 1), PerDiS should support applications much better than a remote-object client-server system. The main focus is on simplicity, ease of use, and adaptation to the needs of the application area.

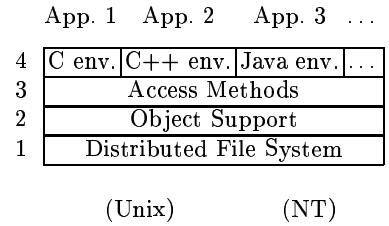

Figure2. Layers of the PerDiS design

3 Design

PerDiS provides the logical layers of functionality shown in Figure 2. PerDiS is hosted on traditional operating systems (OS), currently Unix and NT. Of the host OS, PerDiS uses only the local file service for storing its data, and TCP/IP sockets for communication with remote nodes (a node is a machine participating in a PerDiS platform.) The distributed sharing of data is managed by PerDiS, independent of the host OS.

3.1 Secure Transactional File System

Layer 1 of the PerDiS design provides a secure, distributed file system with transactional semantics [2]. Each cluster is stored in its own file with access restricted to the PerDiS Daemon by OS protection.

Each cluster has a PerDiS specific access control list (ACL), and an application process will gain access to a cluster only if the user presents credentials matching the ACL. Furthermore, a node will classify other nodes as trusted or untrusted. Communication with a trusted node uses a lightweight protocol. When communicating between untrusted nodes, each one double-checks that the other one is doing the right thing; for instance one node will not accept updates from a node that cannot prove ownership of a write lock. More details about the security architecture are in Section 4.5.

The PerDiS platform provides cooperative caching. A cluster can be stored anywhere, and even saved on local disk for fault tolerance and availability. However, it is a requirement, for security and legal reasons, that every cluster has a designated *home site*. The home site is guaranteed to store the most recent, authoritative version of the cluster.

3.2 Object Support

Layer 2 provides support for objects, i.e., collections of contiguous bytes. It is independent of any particular programming language, but it attaches meta-data to an object, for use by the language-specific support of Layer 4 (language-specific runtime service).

Layer 2 knows about the pointers contained in an object; the data is otherwise uninterpreted by this layer. It supports pointer persistence (swizzling), persistence by reachability, and garbage collection.

It is transparent whether a pointer points within the same cluster or into another cluster; however cross-cluster pointers are known to be more costly than intra-cluster ones. Application programmers can control the cost of following pointers by increasing locality within the same cluster.

3.3 Access Methods

Layer 3, also language-independent, provides naming of roots and access to objects in memory. The latter is very similar to what is found in a traditional DSM.

Its `link_root` primitive names a pointer with a URL [4], thus making it a root. Later, an application can enter the system via any root by providing its URL to the `open_root` primitive.

Two alternative memory access methods are provided. The first is based on an explicit API, whereby an application process calls the `hold` primitive to declare intent to operate on some data. As the application navigates through the object graph (starting from a root), it calls `hold` for each object. The arguments to `hold` include the extent of the object and the access mode (e.g., read or write). If the object indicated by `hold` is in a cluster that has not yet been accessed, this layer calls Layer 1 to open the new cluster (thereby checking access rights), thus providing seamless access across cluster boundaries. It calls Layer 1 to perform transactional concurrency control accessing the byte range in the specified mode, to record the access with the transaction manager, and to load the data into memory. It calls Layer 2 (Object Support) to do pointer swizzling, which in turn up-calls the language-specific runtime of Layer 4 to do type-checking.

The second access method, called the "compatibility interface" makes it easy to make existing centralized C or C++ programs distributed and persistent. A program can choose not to explicitly call `hold`. On initial entry via `open_root`, the corresponding cluster is opened and its pages protected against all access. As the application follows a pointer, the operating system might signal a page fault to this layer, which is handled as an implicit `hold` covering the whole faulting page.

3.4 Language-specific Run-time Services

Layer 4 provides language-specific run-time services, such as allocation of typed objects and type-checking procedures. It takes advantage of the hooks provided by Layer 2 (object support) to store the type information for each object. The swizzler, in Layer 2, up-calls this layer at swizzling time to ensure that pointers are correctly typed.

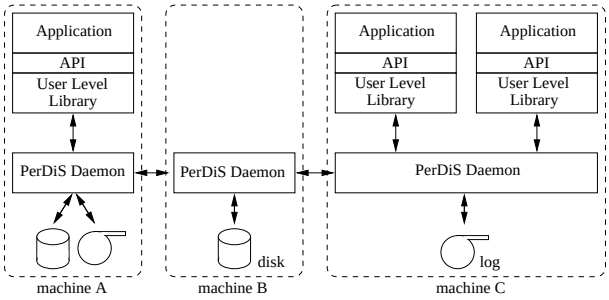

Figure3. PerDiS platform design

4 Architecture and Implementation of the PerDiS Platform

PerDiS is a large, long-term project with ambitious goals. The major portion of the design is available in the prototype platform, although some parts are not implemented yet (see Section 6), most notably swizzling and fault-tolerant caches.

4.1 Structure

The PerDiS architecture is multi-process and peer-to-peer. Figure 3 illustrates the breakdown into processes, which isolates crucial platform functions from applications, and applications from one another, while providing reasonable performance. This breakdown is very much orthogonal to the layered design of Section 3; indeed, a bit of each layer can be found in each process.

A node runs a single *PerDiS Dæmon* (PD), and any number of application processes. Applications interact with PerDiS through an API layer, which interfaces to the *User Level Library* (ULL). A ULL communicates only with its local PD.

The ULL provides memory mapping, transactions, private data and lock caching, swizzling and unswizzling, and the "creative" part of garbage collection (see Section 4.4). When the application needs locks or reads or writes stored data, its ULL makes requests to the local PD.

A PD provides a data and lock cache shared by all applications at this node, maintained coherent with other PDs. It logs the results of transactions. It also contains security modules, and the "destructive" part of garbage collection. A PD communicates with other PDs over the network. They exchange notification messages for locks, updates and garbage collection. They cooperate to locate the home site of a cluster.

To illustrate the responsibilities of ULL and PD, consider a typical application scenario. An application starts a new transaction. This creates an instance of a transaction manager in the ULL, and causes the PD to start a log. Then

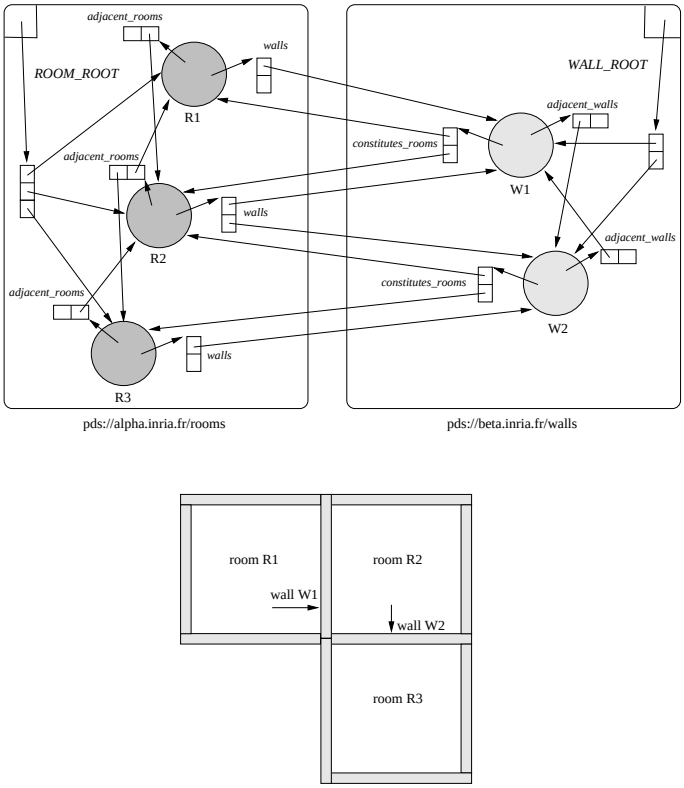

Figure4. Distributed data example: rooms and walls

the application opens a cluster. When the application performs `hold` (either explicitly or through a page fault), and the data is not already cached in the ULL, the ULL requests the corresponding data from the PD. The ULL also requests locks to maintain consistency of its cache with the PD. If the PD does not have some data or lock in its cache, it fetches it from another cache or from the cluster's home site. When the application commits, it sends garbage collection information, updates and locks to the PD.

4.2 API

We will illustrate the PerDiS API through a simple example. Figure 4 displays the data structures, and Figure 5 contains the source code. Suppose that we represent a building by `room` and `wall` objects. Each `room` points to adjacent rooms and to its own walls, and each `wall` points to adjacent walls and to the rooms it encloses. We arbitrarily decide to store all `room` objects in cluster `pds://alpha.inria.fr/rooms` and all `wall` objects in cluster `pds://beta.inria.fr/walls`. The URLs identify the cluster's home site and file name. We give each

```
transaction *t = new_transaction (&standard_pessimistic_transaction);

// open root, navigate
cluster *room_cluster = open_cluster ("pds://alpha.inria.fr/rooms");
list<*room> *rooms = open_root (list<*room>, "ROOM_ROOT",
                                intent_shared, room_cluster);
room *r1 = rooms->first();
room *r2 = rooms->second();

// Change dimensions of wall W1 of R1 and recompute surface and volumes
r1->walls->first()->height = 2.72;
r1->walls->first()->length = 3.14;
r1->walls->first()->compute_surface();
r1->compute_volume();
r2->compute_volume();

// commit, and start another transaction
t = renew_transaction (t, &standard_pessimistic_transaction);

// Add new room R3 that shares wall W2 with R2
room *r3 = pds_new (room, room_cluster) room;

rooms->insert(r3);
r3->adjacent_rooms->insert(r2);
r2->adjacent_rooms->insert(r3);
r3->walls->insert(r2->walls->second());
walls->second()->constitutes_rooms->insert(r3);

end_transaction(t, COMMIT);
```

Figure5. Source code for example

cluster a persistent root (respectively, `ROOM_ROOT` and `WALL_ROOT`) that points to all its objects.

The call to `new_transaction` starts a transaction; the argument requests a pessimistic transaction using the compatibility interface (see Section 3.3). The sequence `open_cluster;open_root` opens the root of the room cluster. The application navigates from the root and changes a wall's dimensions; note that the wall cluster does not need to be opened explicitly. The application recomputes affected surfaces and volumes. The primitive `renew_transaction` commits the current transaction and atomically starts a new one, retaining all its locks, data, and mappings. Then we create a new `room` in `room_cluster`; the macro `pds_new` calls the PerDiS primitive `allocate_in` to reserve space in the cluster, then calls the C++ initialization directive `new`. The new room is inserted into the corresponding data structures. Finally, the program commits. For simplicity, we (incorrectly) neglected to check for errors; for instance `renew_transaction` might fail because the first transaction cannot commit.

This example shows that the PerDiS approach is powerful and elegant. Thanks to caching, all data manipulation is local and distribution is transparent. Local and cross-cluster references are normal pointers. Instead of bothering with distribution, persistence, memory-management, etc., programmers focus on problem solving and application semantics.

Although the argument to `hold` is typically an object or a page, it can in fact be an arbitrary contiguous address range, from a single byte to a whole cluster. Thus, the application can choose to operate at a very fine grain to avoid contention (at the expense of overhead for numerous `hold` calls), or at a very large grain to improve response time (increasing however the probability of false sharing and deadlock, and increasing the amount of data to be logged at commit time). Programmers have full control, if desired, over distribution through the `hold` primitive and through the cluster abstraction.

4.3 Transactions and Caching

An application can request either pessimistic or optimistic concurrency control [21]. It can also request different kinds of locking behaviour, including non-serializable data access, but we will ignore this issue here for the sake of brevity.

A PD caches data and locks accessed by transactions executing at its site. In the current implementation, PDs maintain a sequentially-consistent coherent cache, along the lines of entry consistency [5]. The granularity of coherence is the page.

Transactions run on top of this coherent cache. An application process running a transaction gets a private, in-memory scratch copy of the pages it accesses. An application may update its scratch copy (assuming its write intents were granted). The transaction manager (in the ULL) sends changed data regions to the log (in the PD). When the transaction commits, it flushes any remaining updates, then writes a commit record on the log. The updates are applied to the cache and to disk.

A scratch copy is guaranteed to be coherent when initially copied in from the cache, and again at commit: taking a transactional lock on some datum translates into taking an entry-consistency lock on the corresponding page(s) in the cache. The timing of the entry-consistency locking is different for optimistic and pessimistic transactions. A pessimistic transaction takes a read or write lock as soon as the application issues `hold`, and releases the locks at commit or abort. This blocks any conflicting concurrent transaction. In an optimistic transaction, `hold` reads each page and its *version number* atomically. The transaction takes entry-consistency locks only at commit time; at that time it checks that no version numbers have changed (otherwise the transaction must abort); it performs its updates and releases the locks. Every commit, whether optimistic or pessimistic, increments the version numbers of all pages it modifies. The above guarantees serializable behaviour for both kinds of transactions.

Once the commit record is recorded in the log, the commit is successful. Logged modifications are then applied to the cache, and finally to the files at the clusters' homesites. Our current implementation guarantees the ACID properties

on both the optimistic and pessimistic transaction models. If a PD or a node crashes, a recovery procedure reads the log; transactions whose commit record is on the log are re-done, and those that are still pending are undone. However, caches are not yet fault-tolerant, and a data request sent to a node that has crashed will abort the requesting transaction.

4.4 Garbage Collection

Manual space management, implying potential storage leaks and dangling pointers, is unacceptable in a persistent store. Any leakage would persist and accumulate, overwhelming the store. Dangling pointers (caused by an application program erroneously deleting a reachable object) would make the store unsafe, causing programs to fail unexpectedly, possibly years after the error. In PerDiS instead, storage is managed automatically, using the Larchant distributed garbage collection algorithm [19,20]. Larchant is based on meta-information that supplements pointers, called stubs and scions. A stub describes a pointer that points out of a cluster, and a scion a pointer into a cluster. Stubs and scions are also used by the swizzler.

The algorithm is divided into a "constructive" and a "destructive" part. The constructive part, known as *stub-scion creation* [8], detects a new inter-cluster pointer assigned by the application, creates the corresponding stub, and sends a message to the downstream cluster requesting creation of a scion. Before a transaction's updates are commited, they are analyzed by the stub-scion creation module.

The destructive part, called *garbage reclamation*, runs in the PD. It traces the set of clusters currently cached, using their scions and their persistent roots as starting points for tracing the graph. Objects not visited are unreachable and are deleted. Any given execution of the reclaimer only detects a subset of the actual garbage, but as the contents of caches vary over time, most garbage is deleted with high probability.

The garbage collector must be aware of caching and concurrency, because an object might not be reachable at a particular site but still be reachable globally. It must also carefully order its actions because of possible global race conditions. This is explained in more detail in the Larchant articles [19,20].

4.5 Security

The PerDiS platform protects whole clusters according to user-specified access rights and provides secure communication between PDs. Communication is secured against eavesdropping, impersonation, tampering and replay by attackers from within or outside the VE. The VE is composed of several trust domains. This domain-based trust model enables the encryption and authentication mechanisms to be optimized for the different levels of trust existing within and between organizations.

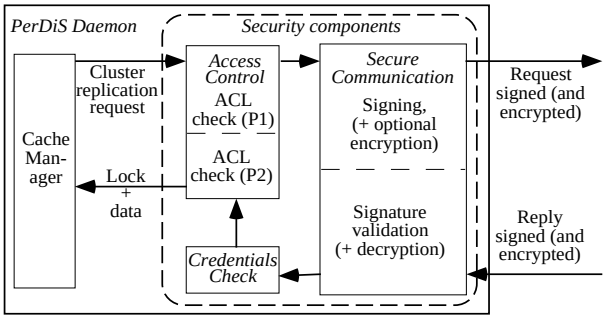

Figure6. Security operations when requesting a cluster copy

Access Control We adopt a role-based model for access control, described in detail in [12]. To summarise: all the cooperative activities carried out by users are assigned to *tasks* and access rights are specified for the roles that users may play within a particular task. A person called a task manager digitally signs *role-in-task* certificates stating which users may play each role. Delegation of roles is supported by the use of task-related delegation certificates. The certificates for each user-level task are stored in a designated cluster and are therefore accessible to the PerDiS security system at participating sites.

Access rights for each cluster are specified in an ACL and access control is applied before a copy is supplied. The access rules are:

- For a PD to obtain a copy of a cluster from another PD the principal behind the requestor must have *read* access rights. This check is applied whenever a PD attempts to acquire a read lock for a cluster and hence get a copy of the data.
- For a PD to obtain a write lock, the principal behind it must have *write* access rights.
  In both cases the access control check is applied at the PD initiating the cluster request in order to give immediate feedback to the application, but because of the lower level of trust between PDs in different domains the check is also applied at the remote PD before granting a lock. This second check can be omitted for PDs in the same domain.
- In addition, a check is needed to ensure that the data accompanying a lock is a genuine version of the relevant cluster. This will be true if the PD granting a lock is backed by a principal that has *write* access rights. (The PD wouldn't have been able to get the lock without those rights). This check is applied by a PD whenever it receives a lock.

Figure 6 shows the security management components within a PD and illustrates their operation to secure a cluster request from the local Cache Manager providing data for an application running on behalf of a principal P1.

After P1's rights have been checked locally, the request is signed, (optionally) encrypted and it is dispatched to the PD currently holding a lock on the required

Table1. Application programming interface. (I, o = input, output parameter. Object = pointer to an object. Intent = read or write. Kind = pessimistic or optimistic. Error codes are omitted)

Cluster management API

	security attribs.	name	cluster
`new`	i	i	o
`open`		i	o
`close`			i

Root management API

	name	cluster	type	object	intent
`link`	i	i		i	
`open`	i	i	i	o	i
`unlink`	i	i			

Data allocation and access API

	cluster	type	object	intent
`allocate`	i	i	o	
`hold`		i	i, o	i

Transaction API

	kind	t'action	status
`new`	i	o	
`renew`	i	i, o	
`end`		i	i

data. The security components at the PD receiving the request validate the request, check P1's credentials and if they allow the access, the second PD returns a signed (and optionally encrypted) reply. Finally, the credentials of the principal behind the second PD are checked at the first PD before the lock and data are passed to the Cache Manager.

Trust management Ultimately, the trust between the participants in a collaborative activity rests on the public keys of the individual users. But to avoid the need for costly bootstrapping of shared keys we establish local trust domains within which a session key is shared between the PDs involved in any PerDiS activity. A trust domain might for example comprise a small organisation or a department of a larger one. The users cooperating on a particular joint task will not generally all be in the same trust domain.

Within a trust domain the assumption of correct PDs on all local computers enables a shared session key to be used and removes the need for the duplication of access control checks described above. But between trust domains the authentication of replication requests requires the use of public key encryption, at least to establish secure channels.

Replication requests may be queued and the request queue may migrate with the lock on a cluster. Hence a PD that sends a replication request cannot be sure that the reply will come from a PD inside the trust domain. We have devised a secure protocol that deals with this issue and is optimised for the local case. Briefly, the sending PD signs (and optionally encrypts) all request messages using the shared session key for the local trust domain. A responding PD in the same trust domain can authenticate the request immediately. If the responding PD turns out to be in another trust domain, it must initiate an additional authentication exchange using the public keys of the two principals involved. The protocol is described in detail in Coulouris et al. [13].

5 Programming with PerDiS

The PerDiS platform supplies application programmers with a whole range of functionalities: object persistence, distribution, caching, transactions, and security. These features are exposed to an application programmer via an explicit API (see Table 1). The interface is *unobtrusive*: very few lines of code are needed to exploit the PerDiS functionality. The PerDiS API consists of four major parts: cluster and persistent root management, data allocation and access, and transactions. More details can be found in the Programmer's Manual and in the PerDiS design documents [18,23].

5.1 API

For cluster manipulation, API functions exist to create a new cluster, to open an existing cluster, and to close an open cluster. Cluster creation requires parameters to specify the security attributes for the cluster, together with its URL.

Since PerDiS' persistence model is based on persistence by reachability, we offer API functions to manage persistence roots. Linking a root associates a name with a pointer. Unlink removes the name; in this case, data reachable only from that root is eventually garbage collected. Opening a root requires, in addition to a name and a cluster, the root object's type. This allows verification of the expected root object type against the actual stored type.

The way data allocation is implemented depends on the programming language. For C and C++ applications we provide alternatives to `malloc` and `new`, respectively. To allocate data in a cluster, one passes a cluster and the type of the object to be allocated. Note that an API for explicit de-allocation of data is deliberately missing, since this is done by the garbage collector only.

All access to the PerDiS store must occur within the context of a transaction. Transactions can be started, terminated and *re-newed*. `Renew` atomically commits and starts a new transaction, retaining the locks, data and mappings of the committed transaction. PerDiS supports different kinds of transactions; starting a transaction requires parameters setting its behaviour. The main parameters specify when and how data locks are taken (as explained in Section 4.3).

5.2 Porting Applications

In general, porting an existing centralized C or C++ application to the PerDiS platform requires data and code conversions, which are both straightforward.

Data conversion requires only small modifications to the application's original I/O modules: (i) create a cluster in place of a file, (ii) perform memory allocation within an appropriate cluster, (iii) create at least one persistent root. These changes can usually be done with very little effort.

Code conversion can be done in several ways, playing on the trade-off between conversion effort and concurrency. We outline the simplest conversion, which takes very little effort (but reduces the level of concurrency): (i) embed the application in a pessimistic transaction that uses the "compatibility interface" (see Section 3.3), (ii) open persistent roots, (iii) replace writes into a file with `renew_transaction` or commit. Again, this involves very few modifications at clearly identifiable places. Thanks to the compatibility mode, data access is trapped using page faults and locks are taken automatically.

These limited modifications bring many gains. In particular, there is no more need for flattening data structures, explicit disk I/O, or explicit memory management. In addition, data distribution, transactions and persistence come for free.

This approach was used to port the applications presented in Section 7.

6 Status

The PerDiS project started in December 1996 and is scheduled as a three-year project. It occupies the equivalent of 8 full-time persons in 6 different institutions across Europe. Time and resources have been obviously insufficient to fully implement the ambitious goals listed above, which however constitute the criteria by which we measure design and implementation decisions.

The source code of the current release is freely available from `http://www.perdis.esprit.ec.org/download/`. It contains some 20,000 lines of C++ and runs on Solaris, Linux, HP/UX and Windows-NT. It is acceptably stable and supports a number of applications, as reported in Section 7.

The platform is intended as a proof-of-concept implementation, and it is not surprising that its performance is not satisfactory yet. Some issues that will be improved are fault tolerant caching, swizzling, type management, performance and elegance.

7 Applications

In this section we present some experiments with applications. These experiments allow us to evaluate the difficulty of building distributed and persistent applications with PerDiS, to compare with other methods and to measure the platform's performance.

Table2. VRML application. For each test set, we provide: (a) Size of SPF file (Kb); number of SPF objects and of polyloop objects. (b) Execution times in seconds: stand-alone centralized version; PerDiS and Corba port, 1 and 2 nodes. (c) Memory occupation in Kbytes (PerDiS 1st column: in memory, 2nd column: in persistent cluster). (d) Number of allocation requests, in thousands (PerDiS 1st column: in memory, 2nd column in persistent cluster)

Test	size	SPF objects	PolyLoops
cstb_1	293	5 200	530
cstb0rdc	633	12 080	1 024
cstb0fon	725	12 930	1 212
demo225	2 031	40 780	4 091

(a) Test applications

Test	Std-Alone	PerDiS 1	Corba 1	PerDiS 2	Corba 2
cstb_1	0.03	1.62	54.52	2.08	59.00
cstb0rdc	0.06	4.04	115.60	4.27	123.82
cstb0fon	0.07	4.04	146.95	5.73	181.96
demo225	0.16	13.90	843.94	271.50	1452.11

(b) Test application execution times (s)

Test	Std-Alone	PerDiS		Corba
		in mem.	pers.	
cstb_1	2 269	2 073	710	26 671
cstb0rdc	2 874	2 401	1 469	51 054
cstb0fon	3 087	2 504	1 759	59 185
demo225				

(c) Test application memory occupation (Kb)

Test	Std-Alone	PerDiS		Corba
		in mem.	pers.	
cstb_1	62	49	14	1 100
cstb0rdc	128	101	29	2 180
cstb0fon	153	121	35	2 543
demo225				

(d) Test application allocations (thousands)

7.1 AP225 to VRML Mapping Application

SPF-AP225 is a standard ASCII file format for representing building elements and their geometry; it is supported by a number of CAD tools. The application presented here reads this format and translates it into VRML (Virtual Reality Modeling Language), to allow a virtual visit to a building project through a VRML navigator.

We chose this application because it is relatively simple, yet representative of the main kernel of a CAD tool. We compare the original, stand-alone centralized version, with a Corba and a PerDiS version.

The stand-alone centralized version has two modules. The *read module* parses the SPF file, and instantiates the corresponding objects in memory. The *mapping module* traverses the object graph to generate a VRML view, according to object geometry (polygons) and semantics. The object graph contains a hierarchy of high-level objects representing projects, buildings, storeys and staircases. A storey contains rooms, walls, openings and floors; these are represented by low-level geometric objects such as polyloops, polygons and points.

In the Corba port, the read module is located in a server which then retains the graph in memory. The mapping module is a client that accesses objects remotely at the server. To reduce the porting effort, only five classes were enabled for remote access: four geometric classes (Point, ListOfPoints, PolyLoop, and ListOfPolyLoops), and one (Ap225SpfFile) allowing the client to load the SPF file and to get the list of polyloops to map. The port took two days. The code to access objects in the mapping module had to be completely rewritten.

In the PerDiS port, the read module runs as a transaction in one process and stores the graph in a cluster. The mapping module runs in another process and opens that cluster. The port took only one day; we used the method outlined in Section 5.2, with no modification of the application architecture. The PerDiS version has the advantage that the object graph is persistent, and it is not necessary to re-parse SPF files each time. The VRML views generated are identical to the original ones.

The stand-alone centralized version is approximately 4,000 lines of C++, in about 100 classes and 20 files. In the Corba version, only 5 of the classes were made remotely accessible, but 500 lines needed to be changed. In the PerDiS version, only 100 lines were changed.

Table 2 compares the three versions for various test sets and in various configurations. Compared to the stand-alone centralized version, performance is low, but this is not surprising for a proof-of-concept platform. Compared to a remote-object system, even a mature industrial product such as Orbix, the PerDiS approach yields much better performance.[3] Memory consumption in the PerDiS version is almost identical to the stand-alone one, whereas the Corba version consumes an order of magnitude more memory.

[3] Note that in Table 2 the PerDiS numbers do not include commit times so that we can compare them fairly with the numbers obtained with CORBA, which were obtained without transactions.

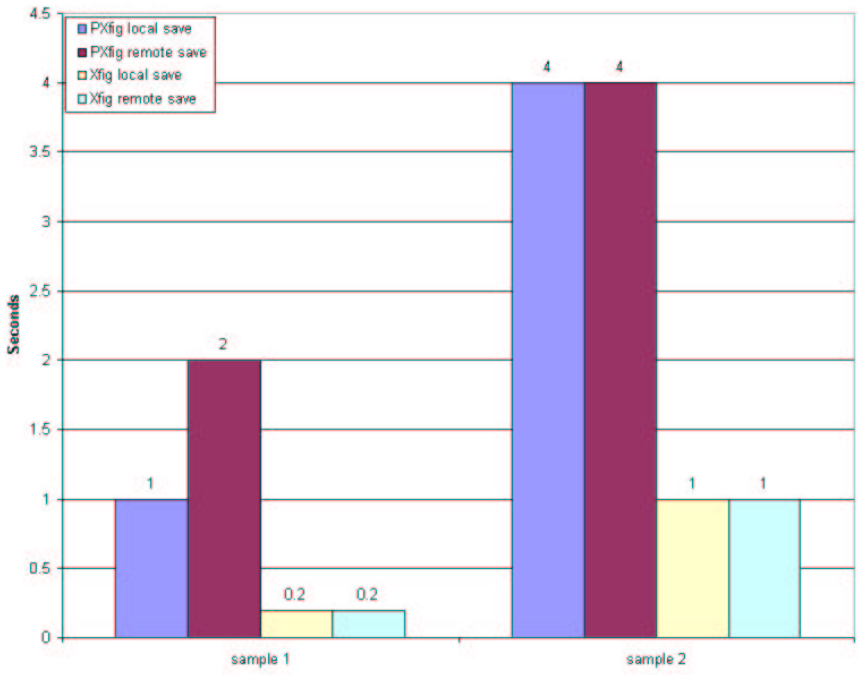

Figure7. PXfig vs. Xfig save time. The figure compares the time to save drawings using PXfig and Xfig (PXfig local/remote = local/remote home site; Xfig local/remote = local file/NFS file)

In Table 2, Std-Alone represents the original, stand-alone application; PerDiS-1 and 2 are the port to PerDiS; both processes run on the same machine, but the cluster's home is either on the same node or on a different one. Corba-1 and 2 are the port to Corba, with the server running on the same machine as the client or on another machine. Size represents the size of the SPF file in kilobytes. Objects is the number of objects in the SPF file, of which Loops represents the number of elementary polyloops. Execution times are in seconds. Allocation sizes are in Kbytes, and allocation requests in thousands. The memory allocation numbers for PerDiS and Corba add up the consumption of both processes.

The one-machine configuration is a Pentium Pro at 200 MHz running Windows-NT 4.0. It has 128 Mbyes of RAM and 100 Mbytes of swap space. In the two-machine configuration for Corba, the server runs on the same machine as above. The client runs on a portable with a Pentium 230 MHz processor, 64 Mbytes RAM and 75 Mbytes swap space, running Windows-NT 4.0. In the two-machine configuration for PerDiS, both processes run on the first machine, whereas its home site is on the second one.

This experience confirms our intuition: the persistent distributed store paradigm performs better (in both time and space) than an industry-standard remote-

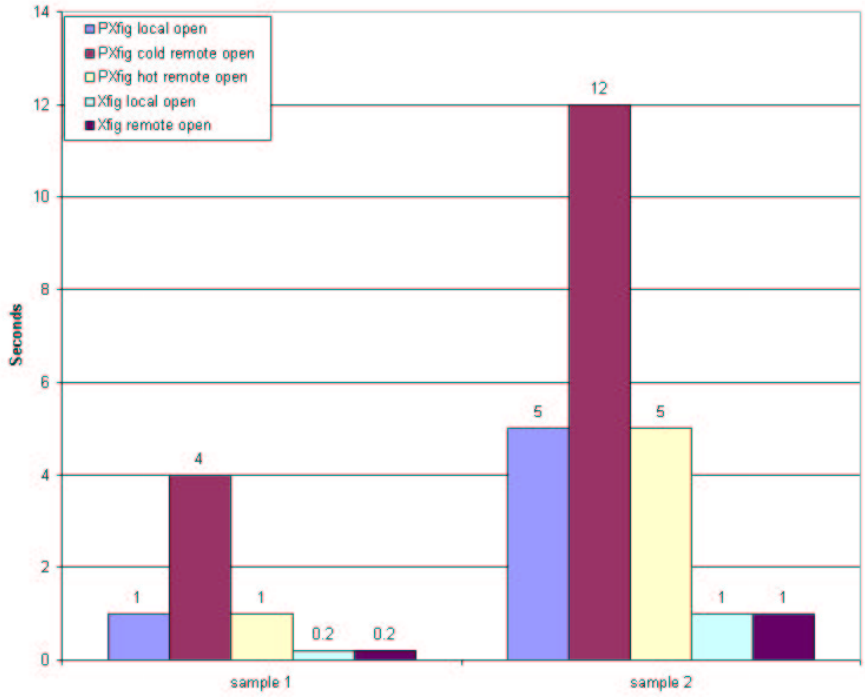

Figure8. PXfig vs. Xfig open time. The figure compares the time to open drawings locally and remotely (PXfig hot/cold = hot/cold PD cache)

invocation system, for data sets and algorithms that are typical of distributed VE applications. It also confirms that porting existing code to PerDiS is straightforward and provides the benefits of sharing, distribution and persistence with very little effort.

7.2 Persistent Xfig

Xfig is a simple drawing tool written in C, freely available on the Internet in source code format. The version ported to PerDiS is Xfig 3.2. We call our port to PerDiS *persistent Xfig (PXfig)*.

We choose Xfig because it is relatively simple and well-written. We expect it to be representative of the behaviour of the class of drawing tools. PXfig allows users to work together on a shared drawing. The port followed the guidelines presented in Section 5.2.

To convert allocation in a transient heap to allocation in a persistent one, 191 lines of code were modified (0.31% of total). 1809 lines were added to provide a graphical interface, e.g. to open a cluster containing a drawing and to commit or abort. We modified 24 source files, out of a total of 141, and we created 8 new

Table3. PXfig vs. Xfig storage requirements. The fields are: number of objects; size of Xfig text file (KB) and size of PerDiS cluster (KB)

	Objects	Xfig file	PXfig cluster
sample 1	4511	83.6	181.9
sample 2	21326	274.4	653.4

source files. PXfig preserves the capability to work with classical Xfig drawings. Consequently, code to parse and flatten Xfig file format remains in PXfig sources; if we removed this backward compatibility, at least 4000 lines of code (6.3% of total) could be deleted.

We present in Figures 7 and 8 a performance comparison between Xfig and PXfig. All the measurements were done on two Sun UltraSparcs at 140 Mhz with 128Mb of memory, running Solaris 2.5 and connected by a 100MB Ethernet. Two complex Xfig drawings, named sample1 and sample2 were used to measure basic operation speed. Table 3 shows the size and number of graphics objects for each sample.

For each test, the measurements were repeated with both the explicit and compatible API but we found no significant performance differences (in all cases we have used only pessimistic transactions). Note also that the results for remote open and remote save for classical Xfig are based on NFS.

The conclusion of this experience is that Xfig was quite easy to port, allowing distributed cooperative edition of Xfig drawings. Note the size increase between Xfig and PXfig files in Table 3, which represents a negative (if not unexpected) consequence of using a binary format. Performance of PXfig is acceptable for an interactive drawing tool. We expect future optimizations of the platform to improve performance significantly.

7.3 Genome Application

LASSAP[4] is an application that searches through a database of genome sequences for a match with a particular pattern. In the original implementation, sequences are stored in a text file, which LASSAP parses at each execution.

We ported LASSAP to PerDiS to see how our system will behave outside the targeted application area. LASSAP should benefit from PerDiS by running multiple searchs in parallel without re-parsing the database every time.

The original version of LASSAP parses the data for a single sequence and stores it into a statically-allocated object. This object is large enough for the biggest possible sequence. After comparing one sequence, the next one overwrites the previous one in the same location.

Porting LASSAP was far harder and less beneficial than expected. For the PerDiS port, we kept the same object type, but we dynamically allocate a new object for each sequence. This in itself required major changes in the code. Since

[4] LArge Scale Sequence compArison Package, `http://www.gene-it.com`

we did not change the object type, and the original is generously sized, a lot of memory is wasted, causing poor performance because of the cost of allocation and of the loss of locality. We don't present any figures here because they do not provide much insight.

This experience teaches us that applications that use overlayed static data structures are not easy to port to PerDiS.

8 Related Work

Given that the distributed sharing of objects is an active research area, PerDiS can be compared to many different kinds of systems.

8.1 Distributed File Systems and DSMs

The differences between PerDiS and a distributed file system (DFS) should now be clear: Layer 1 of PerDiS provides a DFS; its other layers add support for objects, for DSM functionality, and for language-specific functionality.[5]

Differences with a DSM are also clear. DSMs have been most successful in supporting multi-threaded parallel programs, whereas PerDiS facilitates sharing between different programs.

Many of the ideas in PerDiS are direct descendents of the so-called Single-Address Space Operating Systems (SASOS) such as Opal [11], Grasshopper [14], or EOS [22]. The main difference with PerDiS lies in the integration of persistent object systems and object-oriented database concepts, e.g., transactions, persistence by reachability and garbage collection. This is why our system uses transactions and incorporates a security architecture. We also provide persistence by reachability, essential for the long-term safety of the store.

8.2 Object-oriented databases, persistent object systems and object-based systems

Object-oriented databases (OODBs) [35] such as O_2 [15], Thor [25], GemStone [9], or ObjectStore [24] share many of the same goals as PerDiS. OODBs support complex data types, persistently preserving the structure and type of data. However, they are very heavyweight, and often come with their own specialized programming language.

Moreover, an OODB typically manages a database as a tightly encapsulated data unit, residing on a specific server entrusted with crucial functions such as security, recovery and schema enforcement. Transactions can access multiple databases, but this is an infrequent and heavyweight action. In contrast,

[5] Let us emphasize that, in contrast to well-known DFSs such as NFS, AFS, or Sprite, our Layer 1 is a *transactional* and *secure* DFS. Our experience shows that transactions cannot be efficiently layered on top of a standard DFS, and that security permeates the whole system and cannot be added as an afterthought.

the PerDiS shared object world is diffuse, being composed of clusters which are dynamically opened according to application navigation. PerDiS accommodates this world of data *without frontiers* by distributing server functions across cooperative caches, performing schema validation incrementally, and permitting applications to customize their transaction semantics.

As an example, compare PerDiS and ObjectStore. Both provide coherent distributed access to shared objects, exploit client caching, and rely on page fault interception to swizzle pointers. However, ObjectStore is a full-featured system for database management, whereas PerDiS is intended to support object sharing for a diverse range of applications, of which CAD is the primary motivating example.

Persistent object systems such as Mneme [26], Shore [10], Texas [31] and PJama [28] have a similar lightweight approach to PerDiS. However, most of these such systems do not address the issues of distribution and security, they are mostly client-server based, and are limited to traditional transaction models which are too restrictive to the applications we are considering.

Object-based systems such as Comandos [32], SOS [30], or Emerald [7], for example, have tried to simplify the construction of applications that handle persistent and distributed data. However, they do not support replicated data, they are mostly client-server based, objects are acessed by remote invocation, and security has not been fully considered from the beginning.

8.3 Remote Object Systems

Remote object systems such as Corba [16], DCOM [29] or Java RMI [34] let a client invoke objects located on a remote server by Remote Procedure Call (RPC) [6]. RPC solves the problems of identification and remote access. However, every remote data access is burdened with communication to the server, which becomes a performance and availability bottleneck. This makes the client-server architecture inadequate for interactive CAD and cooperative applications, especially upon the WAN connections typical of a VE. In addition, RPC does not support coherence of data viewed by multiple clients. Finally, it imposes an interface definition language to program remote data access, separate from the programming language. Some of the above systems provide transactional and/or persistence services, but they are poorly integrated into the system, being very heavy-weight and awkward to use.

9 Future Work

Concurrent engineering transactions are of long duration; they sometimes need to read data that is being actively modified by another transaction. They are interactive, implying that aborts are perceived as intolerable by users.

In future work, we plan to allow a transaction that would abort under the standard ACID policy, to be committed tentatively until its results are either

reconciled with the store or definitely abandoned. Thus, the work done within a transaction might not be completely lost.

We also have a list of functionalities which are planned for future research. These include support for versioning and reconciliation of write conflicts, sharing data between heterogeneous machine types, schema evolution, and automatic reclustering. We plan to provide each cluster with its own choice of policies, for replication and coherence control, concurrency control, swizzling and garbage collection.

Some open questions remain. For instance, persistence by reachability is the cleanest persistence model, but it is not clear how programmers can make the most effective use of it.

10 Conclusion

We presented PerDiS, a new persistent distributed store providing support for cooperative applications in the framework of a virtual enterprise.

PerDiS automatically manages distribution, persistence, memory management, caching and security in a well defined, integrated, and automatic way. Distributed programming is very simple because PerDiS provides the same memory abstraction as in the centralized case. Although on the one hand the platform provides transparency to programmers who prefer to let the system take care of the difficult issues, it also provides powerful primitives that provide full control to the knowledgeable application programmer.

PerDiS integrates in a novel fashion different techniques, such as distributed shared memory, transactions, security, and distributed garbage collection. This unique combination makes porting of existing centralizaed applications very easy. In addition, these applications have an increased functionality because they can make full use of PerDiS. We also achieve good overall performance because we cache data, taking advantage of the locality characteristics of the application area.

References

1. Virginie Amar. *Intégration des standards STEP et CORBA pour le processus d'ingénierie dans l'entreprise virtuelle.* PhD thesis, Université de Nice Sophia-Antipolis, September 1998.
2. João Garcia, Paulo Ferreira, and Paulo Guedes. The PerDiS FS: A transactional file system for a distributed persistent store. In *Proc. of the 8th ACM SIGOPS European Workshop*, Sintra, (Portugal), September 1998.
3. M. P. Atkinson, P. J. Bailey, K. J. Chisholm, P. W. Cockshott, and R. Morrison. An approach to persistent programming. *The Computer Journal*, 26(4):360–365, 1983.
4. T. Berners-Lee, L. Masinter, and M. McCahill. Uniform resource locators (URLs). Request for Comments 1738, December 1994.
5. B. Bershad, M. J. Zekauskas, and W. A. Sawdon. The Midway distributed shared memory system. In *Proc. of the 1993 CompCon Conf.*, 1993.

6. A. D. Birrell and B. J. Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Programming Languages and Systems*, 2(1), February 1984.
7. A. Black, N. Hutchinson, E. Jul, and H. Levy. Object structure in the Emerald system. In *ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, Portland, Oregon, October 1986.
8. Xavier Blondel, Paulo Ferreira, and Marc Shapiro. Implementing garbage collection in the PerDiS system. In *Int. W. on Persistent Object Systems: Design, Implementation and Use*, Tiburon CA (USA), August 1998.
9. P. Butterwoth, A. Otis, and J. Stein. The GemStone object database management system. *Com. of the ACM*, 34(10):64–77, October 1991.
10. Michael J. Carey, David J. DeWitt, Michael J. Franklin, Nancy E. Hall, Mark L. McAuliffe, Jeffrey F. Naughton, Daniel T. Schuh, Marvin H. Solomon, C. K. Tan, Odysseas G. Tsatalos, Seth J. White, and Michael J. Zwilling. Shoring up persistent applications. In *Proc. Int. Conf. on Management of Data (SIGMOD)*, pages 383–394, Minneapolis MN (USA), May 1994. ACM SIGMOD.
11. J. S. Chase, H. E. Levy, M. J. Feely, and E. D. Lazowska. Sharing and adressing in a single address space system. *ACM Transactions on Computer Systems*, 12(3), November 1994.
12. George Coulouris, Jean Dollimore, and Marcus Roberts. Role and task-based access control in the PerDiS project. In *W. on Role-Based Access Control*, George Mason University, VA (USA), October 1998. ACM.
13. George Coulouris, Jean Dollimore, and Marcus Roberts. Secure communication in non-uniform trust environments. In *ECOOP W. on Dist. Object Security*, Brussels (Belgium), July 1998.
14. Alan Dearle, Rex di Bona, James Farrow, Frans Henskens, Anders Lindström, John Rosenberg, and Francis Vaughan. Grasshopper: An orthogonally persistent operating system. *Computing Systems*, 7(3):289–312, 1994.
15. O. Deux et al. The 0_2 system. *Communications of the ACM*, 34(10):34–48, October 1991.
16. Digital Equipment Corporation, Hewlett-Packard Company, HyperDesk Corporation, NCR Coporation, Object Design, Inc., and SunSoft, Inc. The Common Object Request Broker: Architecture and specification. Technical Report 91-12-1, Object Management Group, Framingham MA (USA), December 1991.
17. Daniel R. Edelson. Smart pointers: They're smart, but they're not pointers. In *C++ Conference*, pages 1–19, Portland, OR (USA), August 1992. Usenix.
18. Marc Shapiro et al. The PerDiS API. `http://www.perdis.esprit.ec.org/deliverables/sources/interfaces/pds_api.h`, May 1997.
19. Paulo Ferreira and Marc Shapiro. Garbage collection and DSM consistency. In *Proc. of the First Symposium on Operating Systems Design and Implementation (OSDI)*, pages 229–241, Monterey CA (USA), November 1994. ACM.
20. Paulo Ferreira and Marc Shapiro. Modelling a distributed cached store for garbage collection. In *Proc. of the 12th European Conf. on Object-Oriented Programming (ECOOP), Brussels (Belgium)*, July 1998.
21. Michael Franklin, Michael Carey, and Miron Livny. Transactional client-server cache consistency: Alternatives and performance. *ACM Transactions on Database Systems*, 22(3):315–363, September 1997.
22. Olivier Gruber and Laurent Amsaleg. Object grouping in EOS. In *Proc. Int. Workshop on Distributed Object Management*, pages 184–201, Edmonton (Canada), August 1992.

23. Sytse Kloosterman and Xavier Blondel. *The PerDiS Reference Manual, version 2.1.* INRIA, B.P. 105, 78150 Le Chenay Cedex, France, 2.1 edition, May 1998. `ftp://ftp.inria.fr/INRIA/Projects/SOR/PERDIS/PLATFORM/PPF-2.1/ppf-2-1-manual.ps.gz`.
24. Charles Lamb, Gordon Landis, Jack Orenstein, and Dan Weinreb. The ObjectStore database system. *Communications of the ACM*, 34(10):50–63, October 1991.
25. Barbara Liskov, Mark Day, and Liuba Shrira. Distributed object management in Thor. In *Proc. Int. Workshop on Distributed Object Management*, pages 1–15, Edmonton (Canada), August 1992.
26. J. Eliot. B. Moss. A performance study of the Mneme persistent object store. *ACM Transactions on Information Systems*, 8(2):103–139, April 1990.
27. J. Eliot B. Moss. Working with persistent objects: To swizzle or not to swizzle. *IEEE Transactions on Software Engineering*, 18(8):657–673, August 1992.
28. Tony Printezis, Malcom Atkinson, Laurent Daynes, Susan Spence, and Pete Bailey. The design of a new persistent object store fir pjama. In *International Workshop on Persistence for Java*, San Francisco Bay Area, California (USA), August 1997.
29. Roger Sessions. *COM and DCOM: Microsoft's Vision for Distributed Objects.* Wiley, December 1998. ISBN 0-471-19381-X.
30. Marc Shapiro, Yvon Gourhant, Sabine Habert, Laurence Mosseri, Michel Ruffin, and Céline Valot. SOS: An object-oriented operating system — assessment and perspectives. *Computing Systems*, 2(4):287–338, December 1989.
31. K. Singhal, S. Kakkad, and P. Wilson. Texas: An efficient, portable persistent store. In *Proc. of the Fifth International Workshop on Persistent Object Systems Design, Implementation and Use*, pages 13–28, San Miniato Pisa (Italy), September 1992.
32. Pedro Sousa, Manuel Sequeira, André Zúquete, Paulo Ferreira, Cristina Lopes, José Pereira, Paulo Guedes, and José Alves Marques. Distribution and persistence in the IK platform: Overview and evaluation. *Computing Systems (Fall 1993)*, 6(4), 1993.
33. Paul R. Wilson and Sheetal V. Kakkad. Pointer swizzling at page fault time: Efficiently and compatibly supporting huge address spaces on standard hardware. In *1992 Int. Workshop on Object Orientation and Operating Systems*, pages 364–377, Dourdan (France), October 1992. IEEE Comp. Society, IEEE Comp. Society Press.
34. Ann Wollrath, Roger Riggs, and Jim Waldo. A distributed object model for the java system. In *Conference on Object-Oriented Technologies*, Toronto Ontario (Canada), 1996. Usenix.
35. S. Zdonik and D. Maier. *Readings in Object-Oriented Database Systems.* Morgan-Kaufman, San Mateo, California (USA), 1990.